A Formal Non–Heuristic ATPG Approach

M. Henftling H. Wittmann K. J. Antreich
Department of Electrical Engineering
Institute of Electronic Design Automation
Technical University of Munich, 80290 Munich, Germany

Abstract

This paper presents a formal approach to test combinational circuits. For the sake of explanation we describe the basic algorithms with the help of the stuck–at fault model. Please note that due to the flexibility of our approach, various fault models can be handled. A highly efficient data structure, represented by an implication graph, provides a straightforward evaluation of all relevant local and global implications. The experimental results illustrate impressively the effectiveness of our approach. All the MCNC benchmark circuits are processed without any aborted fault requiring less CPU–time than state–of–the–art tools.

1 Introduction

Testing of integrated circuits is mandatory in order to produce high–quality components. The quality of the test of VLSI-chips is determined by two opposing factors. On the one hand, the test has to assure fault–free circuits; on the other hand, the time requirement for testing a chip must be low. Therefore, a good test is a small set of circuit stimuli that assures high fault coverage.

A standard way to find good test sets is Automatic Test Pattern Generation (ATPG). For every modeled fault in a given circuit, the ATPG–tool has either to generate a test pattern or to prove the fault untestable. A widely–accepted fault model used by most industrial tools is the single stuck–at fault model.

One of the first approaches to ATPG was the D–algorithm [1]. Since Very Large Scale Integration (VLSI) is state–of–the–art, testing is becoming more and more difficult and costly. In order to cope with these problems, efficient gate–level ATPG–tools like PODEM [2] and FAN [3] have been developed. Based upon these two tools, SOCRATES [4] was the first test pattern generator that succeeded in handling all ISCAS '85–benchmark circuits without aborted faults. Ever growing requirements for testing integrated circuits during the last years resulted in new concepts for test generation. Proving a fault untestable or generating a test pattern for a fault may be seen as a decision or satisfiability problem. The realization that ATPG was a well–known Boolean task resulted in new algebraic algorithms for test generation [5, 6, 7].

This paper describes a new and formal approach to test pattern generation for stuck–at faults in combinational circuits. It achieves superior results without any circuit or fault specific heuristic. The approach is based on a set of clauses representing the circuit under test. We handle the set of clauses in a homogeneous way, different than [6]. All clauses are transformed into a data structure called implication graph. Then the implications are performed on this graph which is well suited for test pattern generation. The separation of the part that solves the decision problem from all fault model specific parts guarantees a very flexible test pattern generator that can easily be extended to other fault models [19].

This paper is organized in the following way. The next section reviews the most important techniques of test pattern generation. Section 3 introduces the implication graph as a data structure tailored to perform implications. Section 4 presents the basic ideas in applying the new algebraic algorithm. In Section 5 convincing experimental results demonstrate the effectiveness of our approach. All circuits of the ISCAS '85–benchmarks [9] are treated without any aborted faults. Furthermore, we had no aborted faults for the combinational parts of the ISCAS '89 [10] and MCNC '93–benchmarks [11]. A comparison with various state–of–the–art tools shows that the new approach provides a clear speed–up. Section 6 concludes the paper.

2 State-of-the-Art

Test generation for stuck–at faults is a well studied topic. Many techniques and heuristics have been proposed during the past 20 years; the most important ones are described below.

- The application of a 9 or 15-valued logic [12, 13] instead of the 5-valued logic [1] allows both more precise signal assignments and a significant reduction of the search space.
- During "Learning" [4], dependencies between signals are identified. Exploiting these implications, contradicting signal assignments may be identified at an earlier stage. Therefore, the search space is reduced. This results in accelerated test generation for faults that are hard–to–detect. For faults that are still aborted, it is possible to increase the number of learned implications by considering the current signal assignments. This process is called "Dynamic Learning" [14].
- The idea of "Recursive Learning" [15] is similar to the idea of dynamic learning. Recursive learning, however, ensures that all existing dynamic implications are learned and that all possible signal assignments are made. Consequently, inconsistent signal assignments can be avoided.
- Test pattern generators that are based on a branch–and–bound technique assign logical values to signals.

The size of the search space depends significantly on the order of selecting the signals. The multiple backtrace procedure [16] is a method to select both the signal and the logical value.

- An alternative to the branch–and–bound technique is the application of BDDs, e.g., [5]. A BDD is a data structure that is able to efficiently represent large Boolean functions. Therefore, it is possible to represent all alternatives simultaneously. There are circuits, however, that cannot be represented by a BDD because of exponentially growing memory requirements.

Due to the ever increasing circuit speed, new requirements and fault models have been proposed for test generation. This results in great efforts for test preparation by state–of–the–art tools since

- a complete new test pattern generator has to be developed, or
- existing (stuck–at) test pattern generators have to be rebuilt, and
- valuable techniques that are effective for the old fault model have to be replaced.

This forms the motivation for us to work on the well–studied topic of test generation for combinational circuits.

The result is a test pattern generation kernel. Contrary to former approaches, its advantages are:

- A formal and general description of the test generation problem.
- Efficient development of test generators for various fault models due to the flexibility of our tool. This is enabled by the separation of the fault model independent and the fault model dependent part of the program.
- Extraction of the general and most important ATPG techniques and the avoidance of any fault or circuit specific heuristic.
- A new and highly efficient data structure for implication independent of the used fault model. With the help of this data structure we evaluate all necessary and relevant (local and global) implications.
- Reduction in the number of lines of code up to 80%.
- Realization of new techniques, e.g., bit–parallel test pattern generation and a consistent fault propagation.
- High quality and fast test pattern generation. All the MCNC benchmark circuits are treated without any aborted fault requiring less CPU–time than state–of–the–art tools.

3 The Implication Graph

Performing implications is one of the most important and time consuming tasks of test pattern generation [17]. Therefore, this section will focus on this problem. It will be shown how to derive a data structure [18] from a logic and how to exploit it for fast implication. For the sake of simplification, the necessary steps will be shown for a three valued logic and an AND gate. The extension to arbitrary logics and gates is given in [19].

Let us assume a two input AND gate with the input signals a and b and the output signal c. The function of this gate can be written as

$$c \leftrightarrow (a \wedge b).$$

(The XNOR operation is abbreviated by $\leftrightarrow$.) Exploiting the transformation

$$c \leftrightarrow (a \wedge b) \Leftrightarrow (a \vee \neg c) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg b \vee c)$$

two clauses with two literals and one clause with three literals can be used to describe the logical behavior of the gate and to assure a consistent assignment. Now each

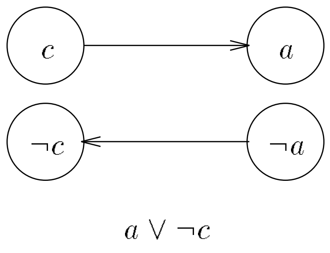

Figure 1: Clauses with two literals

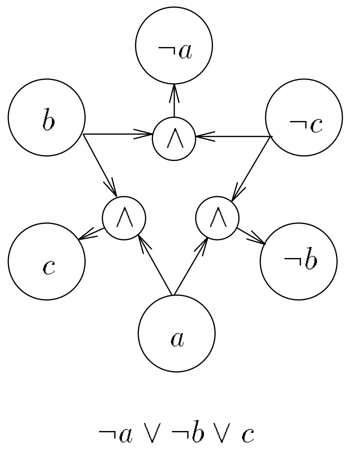

Figure 2: Clauses with three literals

clause with two literals is represented as shown in Figure 1, after transforming

$$a \vee b \Leftrightarrow (\neg a \rightarrow b) \wedge (\neg b \rightarrow a);$$

and each clause with three literals is represented as given in Figure 2.

This procedure leads to an implication graph representation. For each signal, $2 \cdot \lceil log_3 n \rceil$ (n is the number of logical values) nodes are introduced. In case of the 3–valued logic and a signal x, the nodes x and $\neg x$ are introduced. The interpretation is given in Table 1.

To imply from a signal means that the corresponding node is marked. Then all successors are checked if they can be marked. That is the case if

- the node is no $\wedge$–node (see Figure 2) and one predecessor is marked.
- the node is a $\wedge$–node and all predecessors are marked.

These rules are applied until no further propagation of marks is possible.

x	$\neg x$	interpretation	abbr.
no mark	no mark	value unknown	X
no mark	marked	signal is 0	0
marked	no mark	signal is 1	1
marked	marked	conflict	–

Table 1: 3-valued logic representation

This graph is implemented straightforward as a data structure. All local and global implications can easily be computed. The global implications are learned on the graph. The usage of this data structure and its application to perform bit–parallel implication and justification is explained in [8].

4 The ATPG Approach

Our test pattern generator TIP is based on the implication procedure explained in Section 3. In addition to these algorithms there is a need for some fault model specific functions. These functions contain the complete set of constraints that have to be considered for the selected fault model. Please note that the complexity of these functions is linear in the number of gates in case of justification problems. Since there are different fault models that are supported by TIP, we select one of them to explain the principle of the approach: the stuck–at fault model.

The general approach is shown by the flow–chart in Figure 3. The basic steps and the main ideas of the approach will be explained below.

After selecting a fault, the necessary constraints for detecting this fault are performed. The fault is activated and the path in the fanout–free region is sensitized according to the condition of column 3 in Table 2. The first two columns give the type of the gate that should be sensitized and the fault effect at the input of the gate. The entries are explained in Table 3.

Then, the implication procedure that was introduced in Section 3 is performed. If a conflict occurs, the fault is redundant since only necessary conditions to detect the fault are exploited, i.e., no decision was made. There are two

situation		condition	
gate	fault type	FFR	else
AND, NAND	D	1	G1
AND, NAND	$\overline{D}$	1	F1
OR, NOR	D	0	F0
OR, NOR	$\overline{D}$	0	G0
XOR, XNOR	D	–	E
XOR, XNOR	$\overline{D}$	–	E
INV, BUF	D	–	–
INV, BUF	$\overline{D}$	–	–

Table 2: Sensitization conditions

value	fault-free circuit	faulty circuit
D	1	0
$\overline{D}$	0	1
0	0	0
1	1	1
G0	0	x
G1	1	x
F0	x	0
F1	x	1
E	$\substack{0\\1}$	$\substack{0\\1}$

Table 3: Logical values

ways to continue the generation process. Either the logic values that are not satisfied by the values of the primary inputs are justified or the fault effect is propagated. We found out that the first alternative is, in most cases, the better one. There are two reasons for this fact. First, about 90% of all redundant faults can be identified this way. Second, the resulting assignments to the primary inputs form a test pattern that is able to detect the underlying or other untreated faults in about 80% of the cases. The resulting pattern is given to a fault simulator to decide whether the selected fault is detected. If it is not, the fault has to be propagated to one of the primary outputs. For this reason one path starting from the fanout stem of the fanout–free region that contains the fault is selected. This path is sensitized according to the condition of column 4 in Table 2. Then the logic values that are unsatisfied are justified. If no conflict occurs a test pattern for the selected fault has been detected. Otherwise another path is selected, sensitized, and justified. Please note: this type of single path sensitization implicitly considers all multi–path propagation possibilities, that contain the selected path [20]. Hence, the fault is proven to be redundant if no further propagation path exists.

Besides the 9–valued logic and the global implications, TIP does not use any of the other techniques presented in Section 2. It exploits the fast bit–parallel implication procedure [8] based on the implication graph and the separation of the approach into two parts. A part that performs the justification and a part that ensures constraints of the underlying fault model.

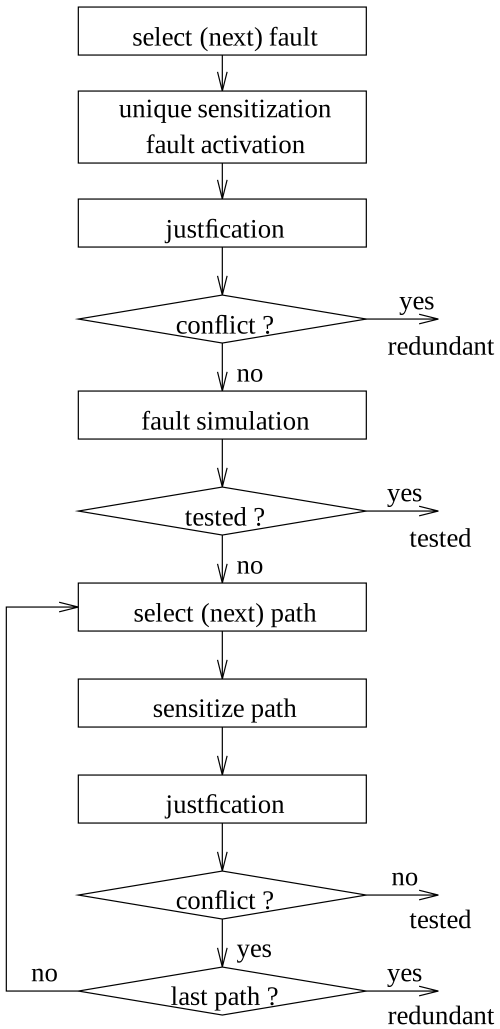

Figure 3: Pseudo-code of TIP/stuck–at

5 Experimental Results

The algorithm of Section 4 was implemented in a C language program, TIP. The results were achieved on a DECstation 5000/200.

Out of the various approaches that treat stuck–at test pattern generation we selected SOCRATES [4] for comparison because it is a well–known structural approach. Moreover, we compare TIP with Nemesis [6] and TRAN [7], two other formal approaches. The comparison with further approaches namely TSUNAMI [5], FAN [16], EST [21], COMPACTEST [22], CONTEST [23], and the approaches of [15] and [24] yields similar results.

Table 4 shows the TIP results for the ISCAS ’85 [9] benchmark circuits and four industrial circuits (their sizes range from 196 to 11629 gates). The next five columns display the number of tested faults, the number of redundant faults, the number of aborted faults, the CPU-time in seconds for test pattern generation and fault simulation, and the number of test patterns after test set compaction using reverse fault simulation. Since we use a comparatively simple fault simulator, the CPU–time additionally necessary for test set compaction is about 35 % of the CPU–time for test generation. The last two columns denote the number of backtracks, and the memory in megabytes that is necessary to build the implication graph. Please note that there is an additional memory requirement especially to store the net list, the test patterns, and the ATPG state for backtracking. The number of aborted faults is zero for all ISCAS ’85, ISCAS ’89 and Addendum ’93 benchmark circuits.

circuit	test	red	ab	time	pat	bt	MB
c432	520	4	0	0.6	54	1287	0.1
c499	750	8	0	0.7	56	4	0.1
c880	942	0	0	0.4	57	0	0.2
c1355	1566	8	0	1.6	87	216	0.3
c1908	1870	9	0	4.6	124	24	0.5
c2670	2630	117	0	6.7	118	113	0.6
c3540	3291	137	0	11.5	162	188	0.8
c5315	5291	59	0	6.7	123	567	1.3
c6288	7710	34	0	3.0	32	0	1.1
c7552	7419	131	0	22.8	222	602	1.6
s35932	34470	3984	0	314.2	68	33k	7.1
s38417	30859	165	0	307.5	911	264	8.7
s38584	34493	1506	0	299.8	631	845	8.7
ind1	390	0	0	0.1	24	0	0.1
ind2	6812	87	0	12.1	182	392	1.6
ind3	11560	210	0	27.2	165	796	2.6
ind4	15237	205	12	3442.1	304	423k	4.5

Table 4: Results of TIP

See Table 5 for a comparison of these results with previously reported stuck–at test pattern generator performances. The first column gives the circuit name. The next four columns show the CPU–times for complete ATPG for SOCRATES (DECstation 5000/200), Nemesis (Sparc 1+) and TRAN (SunSparc2), and our approach. To consider the different machine speed, the third row gives the SpecInt89 of the different machines. The data in Table 5 demonstrates clearly that for all ISCAS ’85 benchmark circuits TIP is significantly faster than all other approaches. It is five to nine times faster than the formal approaches and up to four times faster than the fastest ATPG approach. This result is astonishing since TIP is a formal ATPG tool, which is not specialized in stuck–at faults. TIP is also capable of generating robust and non–robust test patterns for path delay faults assuming enhanced or single scan design [8]. The part that is specially written for stuck–at fault ATPG consists of less than 1000 lines of code. Most of them are necessary to handle the fault list and to initialize the test pattern generation kernel. Please note that the CPU-times of Nemesis and TRAN are reduced by exploiting a random test generation phase first.

TIP is able to generate test patterns for all testable faults; and is able to prove all redundant faults redundant for all ISCAS ’85 [9], ISCAS ’89 [10] and Addendum ’93 [11]

circuit	atpg + fsim time [s]			
	SOCR.	Nemesis	TRAN	TIP
	19.1	11.2	21.7	19.1
c432	1.4	8.5	0.8	0.6
c499	2.9	3.9	1.8	0.7
c880	1.6	37.5	2.9	0.4
c1355	6.1	22.0	6.6	1.6
c1908	12.6	69.8	12.6	4.6
c2670	12.9	371.2	92.9	6.7
c3540	23.6	264.7	23.9	11.5
c5315	14.2	74.8	32.1	6.7
c6288	16.2	147.6	38.0	3.0
c7552	45.1	752.6	298.4	22.8

Table 5: Comparison with other approaches

benchmark circuits. As far as we know, TIP is the first published ATPG tool that achieves this goal with only one test generation phase in a reasonable amount of time.

name	pats	flts	time	$\frac{\text{time}}{\text{fault}}$	norm.
TIP	992	547	18.0	0.033	1.0
Nemesis	297	628	729.9	1.162	20.6
TRAN	n.a.	525	250.3	0.477	16.4
Socrates	1344	527	30.3	0.058	1.8

Table 6: ATPG results for c7552

In another experiment we compared the speed of the different approaches for hard–to–detect faults. For that purpose we performed a random test pattern generation phase to drop the easy–to–detect faults. Then we performed a deterministic test pattern generation phase in order to generate test patterns for detectable but still undetected faults and to identify the redundant faults. Table 6 shows the results of this experiment. We selected the benchmark circuit c7552 because it is known to contain many hard–to–detect faults. Column two gives the number of random patterns. The next two columns present the number of undetected faults after the random pattern generation phase and the CPU–time in seconds that was necessary to classify these faults. Column five and six show the time per fault for deterministic ATPG and CPU–time per fault normalized to TIP and the speed of the machine.

6 Conclusion

We have presented a new and general algorithm for test pattern generation and a method for the propagation of fault effects that works without any heuristics for redundancy detection. The effectiveness of this approach was demonstrated by comparing the results for stuck–at test pattern generation with the most important published methods. It turns out that our approach is up to four times faster than the fastest approaches reported so far. Moreover, it is the first published single phase approach that does not yield aborted faults for any ISCAS ’85 or ’89 benchmark circuit within a reasonable amount of time.

References

[1] J. Paul Roth. Diagnosis of Automata Failures: A Calculus and a Method. *IBM Journal of Research and Development*, pages 278–281, July 1966.

[2] Prabhakar Goel and Barry Rosales. PODEM–X: An Automatic Test Generation System for VLSI Logic Structures. In *Proceedings IEEE/ACM Design Automation Conference*, pages 260–268, June 1981.

[3] Hideo Fujiwara. *Logic Testing and Design for Testability*. Computer Systems Series. The MIT Press, Cambridge, 1985.

[4] Michael Schulz, Erwin Trischler, and Thomas M. Sarfert. SOCRATES: A Highly Efficient Automatic Test Pattern Generation System. In *Proceedings IEEE International Test Conference*, pages 1016–1025, September 1987.

[5] T. Stanion and D. Bhattacharya. TSUNAMI: A Path Oriented Scheme for Algebraic Test Generation. In *Proceedings IEEE International Symposium on Fault–Tolerant Computing*, pages 36 – 43, 1991.

[6] Tracy Larrabee. Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer–Aided Design*, pages 4–15, January 1992.

[7] Srimat T. Chakradar, Vishvani D. Agrawal, and Steven G. Rothweiler. A Transitive Closure Algorithm for Test Generation. *IEEE Transactions on Computer–Aided Design of Integrated Circuits*, pages 1015–1028, 1993.

[8] M. Henftling and H. Wittmann. A Bit Parallel ATPG Approach for Path Delay Faults. In *Proceedings European Design and Test Conference*, pages 521–525, March 1995.

[9] Franc Brglez and Hideo Fujiwara. A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran. In *IEEE International Symposium on Circuits and Systems; Special Session S6AB on ATPG and Fault Simulation*, pages 663–698, June 1985.

[10] Franc Brglez, David Bryan, and Krzysztof Koź mińs ki. Combinational Profiles of Sequential Benchmark Circuits. In *Proceedings IEEE International Symposium on Circuits and Systems*, pages 1929–1934, May 1989.

[11] Franc Brglez. ACM/SIGDA Benchmark Electronic Newsletter DAC ’93 Edition. Microelectronics Center of North Carolina (MCNC), June 1993.

[12] Ch. Cha, W. Donath, and F. Ozguner. 9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits. In *IEEE Transactions on Computer–Aided Design of Integrated Circuits*, March 1978.

[13] S.B. Akers. A Logic System for Fault Test Generation. In *IEEE Transactions on Computers*, pages 620–630, June 1976.

[14] Michael H. Schulz and Elisabeth Auth. Advanced Automatic Test Pattern Generation and Redundancy Identification Techniques. In *Proceedings IEEE International Symposium on Fault–Tolerant Computing*, pages 30–35, June 1988.

[15] Wolfang Kunz and Dhiraj K. Pradhan. Accelerated Dynamic Learning for Test Pattern Generation in Combinational Circuits. In *IEEE Transactions on Computer–Aided Design of Integrated Circuits*, page 684, 1993.

[16] Hideo Fujiwara. FAN: A Fanout-Oriented Test Pattern Generation Algorithm. In *Proceedings IEEE International Symposium on Circuits and Systems*, pages 671–674, June 1985.

[17] S.T. Chakradar, V.D. Agrawal, and M.L. Bushnell. *Neural Models and Algorithms for Digital Testing*. Kluwer Academic Publishers, Boston/ Dordrecht/ London, 1991.

[18] W. Brauer. *Net Theory and Applications*. Lecture Notes in Computer Science No. 84. Springer–Verlag, Heidelberg, 1980.

[19] M. Henftling and H. C. Wittmann. A New Data Structure to Solve the Satisfiability Porblem in Digital Circuits. *Archiv für Elektronik und Übertragungstechnik*, pages 29–43, January 1995.

[20] M. Henftling, H. C. Wittmann, and K. J. Antreich. A Single-Path-Oriented Fault-Effect Propagation in Digital Circuits Considering Multiple–Path Sensitization. submitted for publication to *IEEE/ACM International Conference on Computer–Aided Design*, 1995.

[21] John Giraldi and Michael L. Bushnell. EST: The New Frontier in Automatic Test–Pattern Generation. In *Proceedings IEEE/ACM Design Automation Conference*, pages 667–672, June 1990.

[22] Irith Pomeranz, Lakshmi N. Reddy, and Sudhakar M. Reddy. COMPACTEST: A Method to Generate Compact Test Sets for Combinational Circuits. In *Proceedings IEEE International Test Conference*, pages 194–203, October 1991.

[23] Udo Mahlstedt, Torsten Grüning, Cengiz Özcan, and Wilfried Daehn. CONTEST: A Fast ATPG Tool for Very Lerge Combinational Circuits. In *Proceedings IEEE/ACM International Conference on Computer–Aided Design*, pages 222–225, November 1990.

[24] Sandip Kundu, Leendert M. Huisman, Indira Nair, and Vijay Iyengar. A Small Test Generator for Large Designs. In *Proceedings IEEE International Test Conference*, pages 30–40, September 1992.